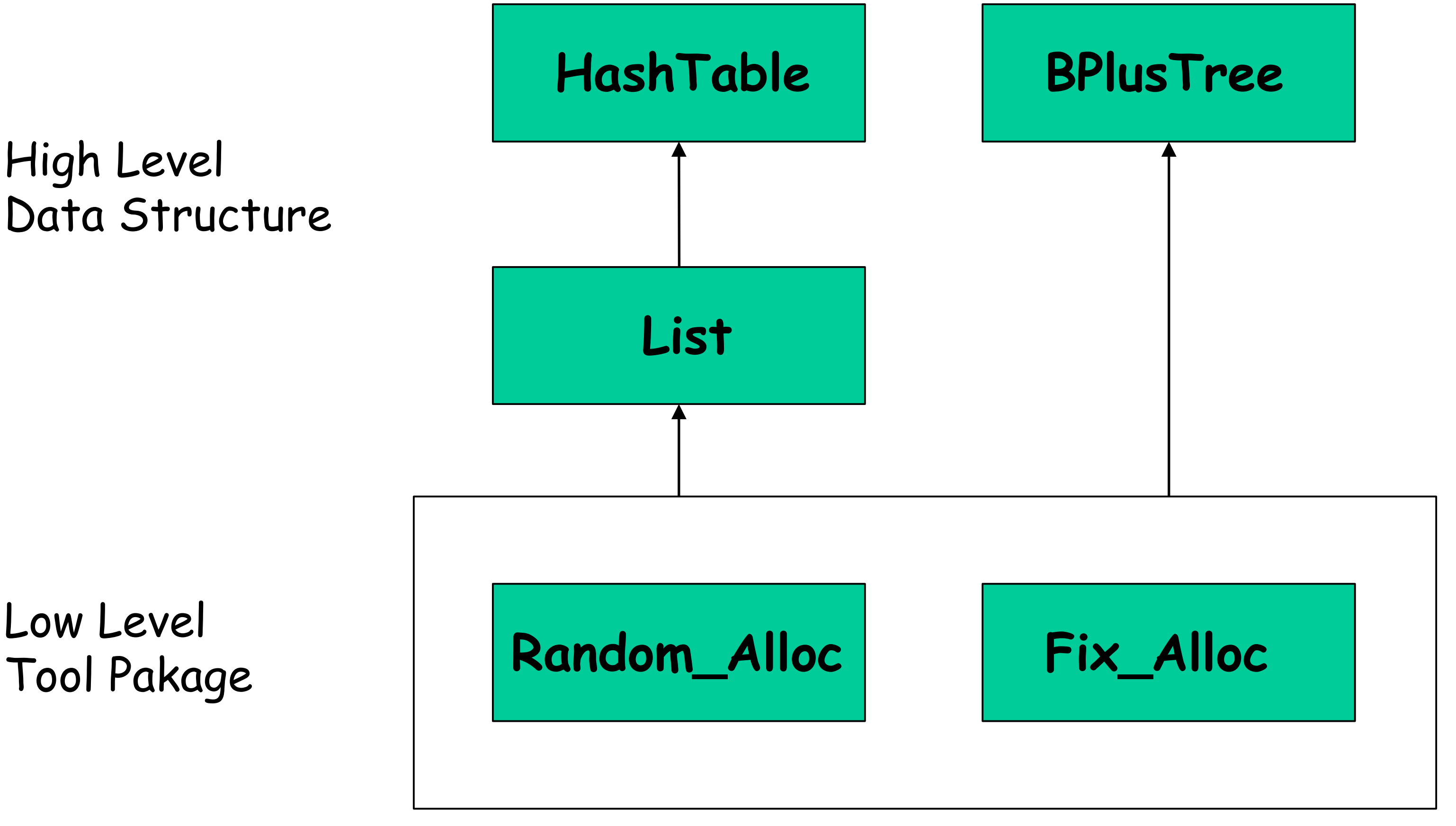
Simple DBMS
Implementation & Comparison

Zhang Yunhao

Outline

- Overview
- Allocation on disks
- Higher level data sturctures
  - List
  - Hash Table
  - B+ Tree
- Testing & Comparison
- Conclusion

Overview


High Level
Data Structure
HashTable
BPlusTree
List
Low Level
Tool Pakage
Random_Alloc
Fix_Alloc

Low Level Tool Package

- Random_Alloc
  - managing Data File
  - insert, remove, query Random length c style strings
- Fix_Alloc
  - managing Index File
  - insert, remove struct/class in disk
  - map, unmap struct/class in VM

Random_Alloc

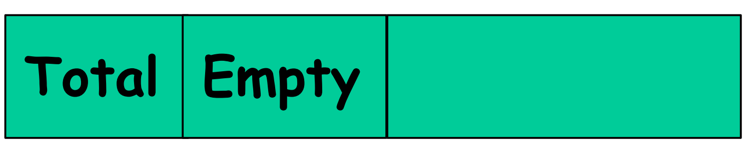
- Public interface:
- class Random_Alloc{

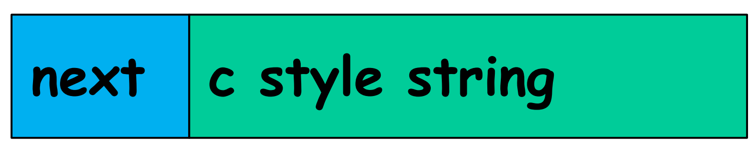
```
public:
    page_t insert(const char *value, int len);
    void remove(page_t addr);
    void query(page_t addr, char* buf);
    bool compare(page_t addr, const char *str);
}
```

Random_Alloc

```
struct Header{
    page_t total;
    page_t empty;
} *header;
```

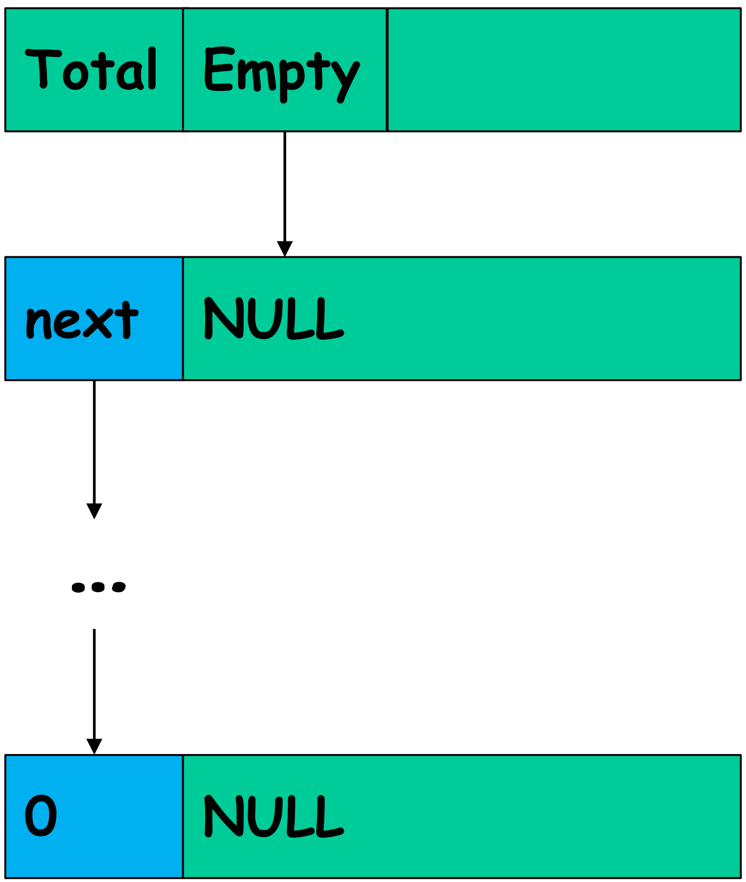
```
struct Page{
    page_t next;
    char value[RAND_BLOCK_SIZE];
};
```





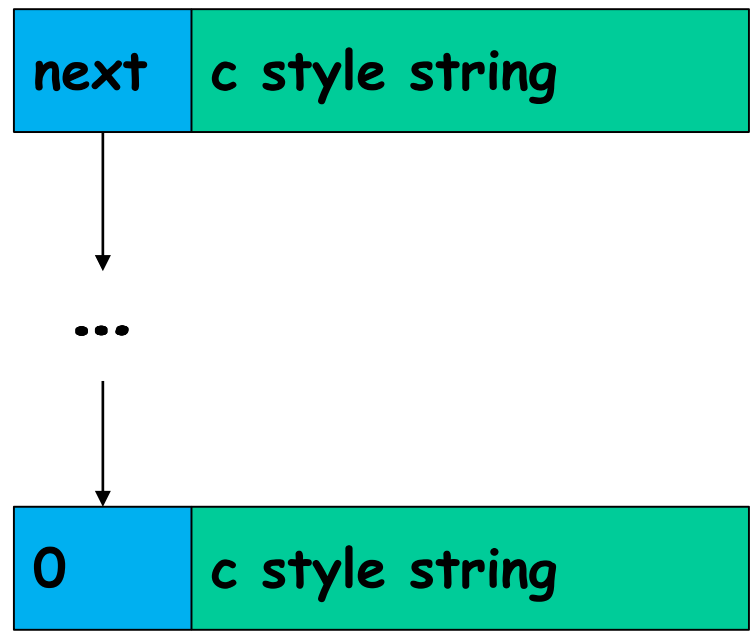
Random_Alloc

Header:



page_t i

Page i:



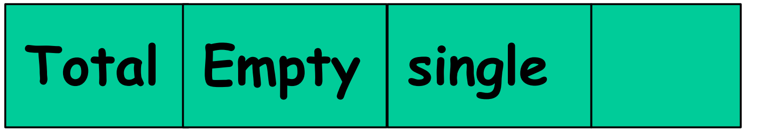
Fix_Alloc

- Public interface:
- class Fix_Alloc{

```
public:
    addr_t insert(void * item);
    void remove(addr_t addr);
    void* use(addr_t addr);
    void unuse(addr_t addr);
}
```

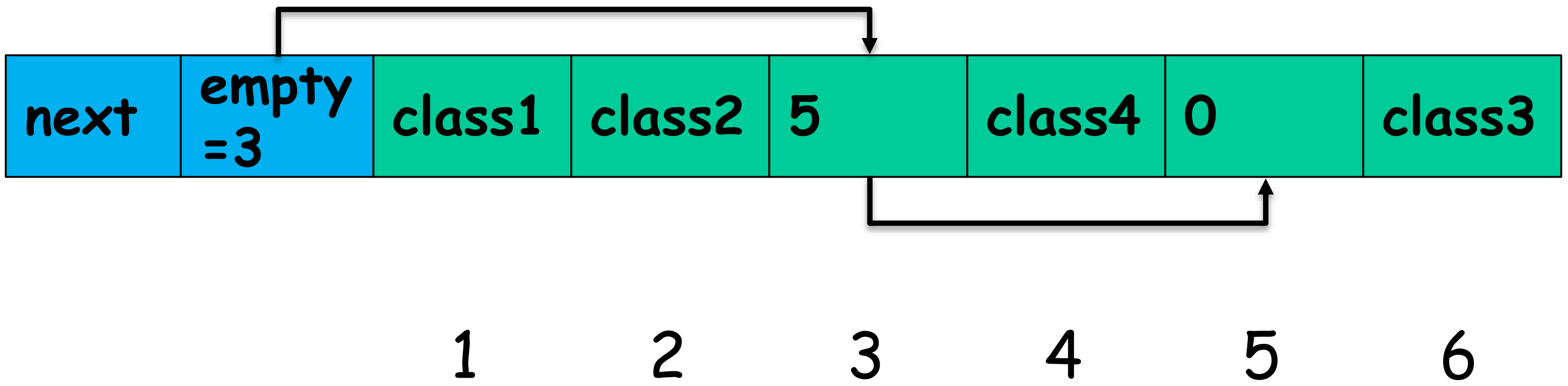
Fix_Alloc

```
struct Header{
        page_t total;
        page_t empty;
        int single;
    } *header
struct Page{
        page_t next;
        int empty;
        char value[FIX_BLOCK_SIZE];
    };
```



Fix_Alloc

Single Page:



Higher Level DS

- Random_Alloc and Fix_Alloc gives interfaces of disk which is similar to main memory
- Easier to implement List, HashTable, B+ Tree

Higher Level DS

- Example:

  How to store, use and remove a struct/class in disk just like in main memory?

```
struct ListEntry{
    addr_t next;
    page_t keyRef, valRef;
};
```

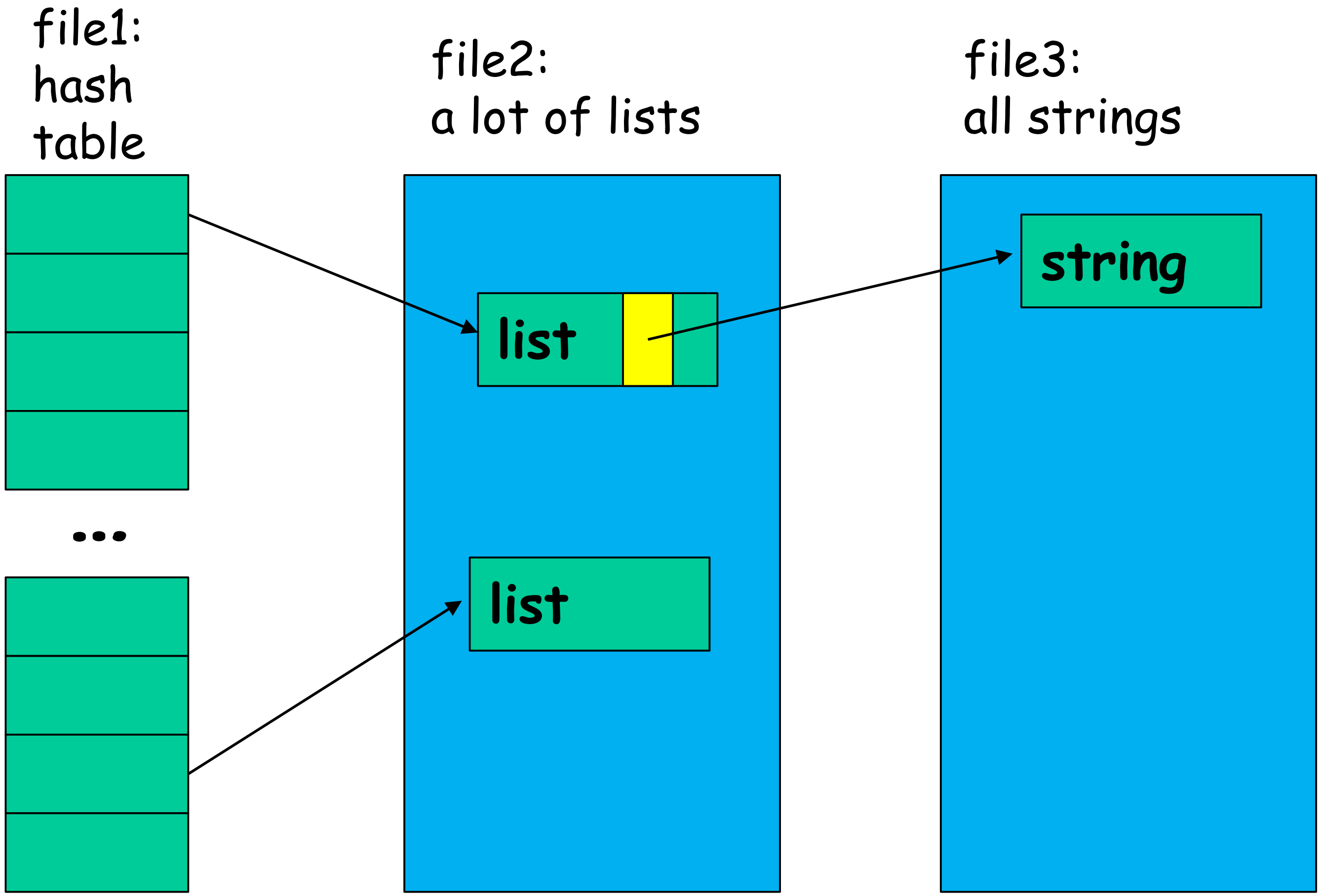
Higher Level DS

- Example in "List.h":

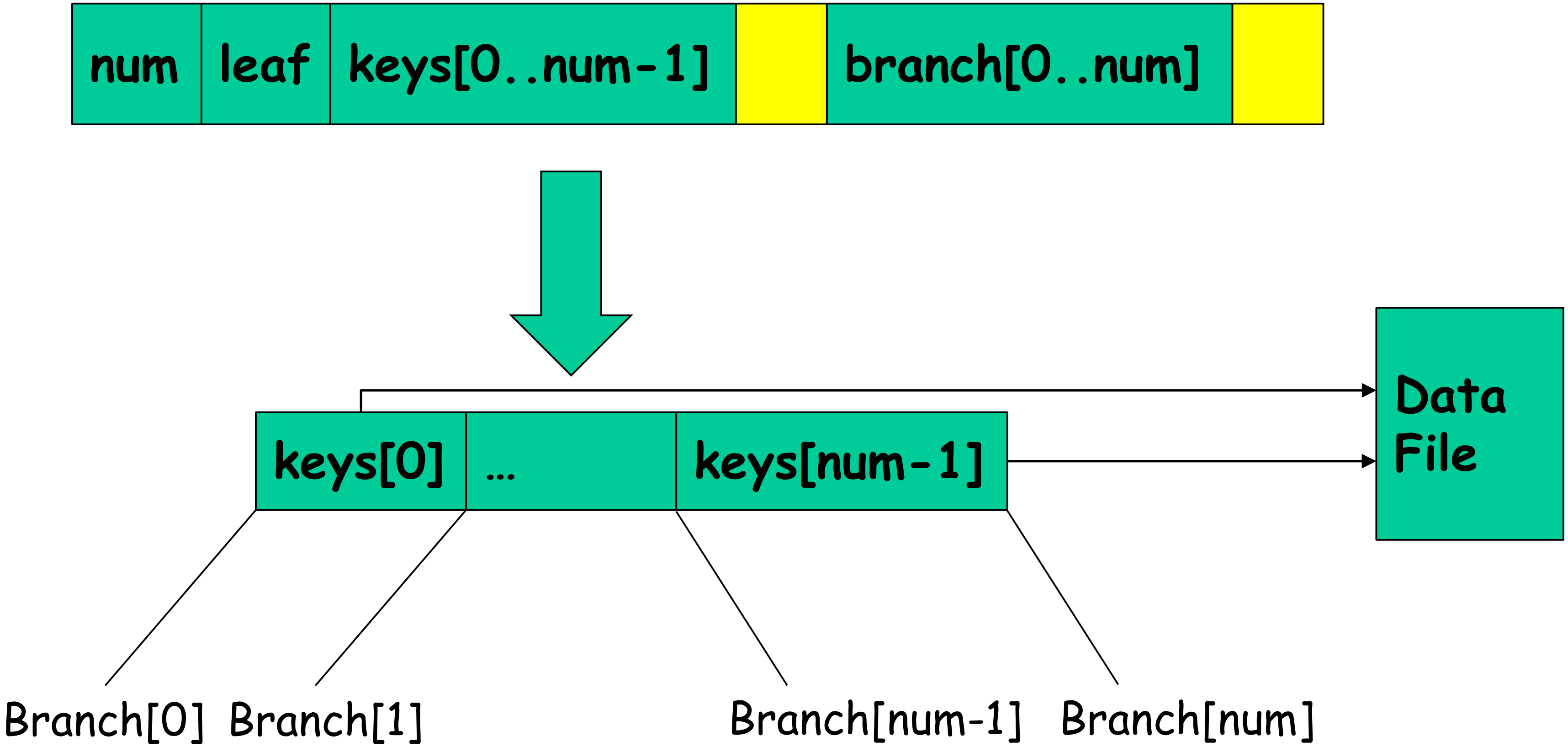
```
Fix_Alloc keySrc;
Random_Alloc valSrc;

ListEntry *ent = (ListEntry*) keySrc->use(now);
ent->valRef = valSrc->insert(value, strlen(value));
keySrc->unuse(now);
```

Hash Table


file1:
hash
table
...
file2:
a lot of lists
list
list
file3:
all strings
string

B+ Tree



num
leaf
keys[0..num-1]
branch[0..num]
keys[0]
...
keys[num-1]
Data File
Branch[0]
Branch[1]
Branch[num-1]
Branch[num]

Test

- test.cpp
- Test File Format
  - N // N lines below
  - 1 key value // insert (key, value)
  - 2 key // remove key
  - 3 key value // (key, old) -> (key, value)
  - 4 key // query key

Test

- Sample input:

```
20
1 U WB8ARp
4 U
1 4Lxj ot
2 U
1 5 gkEw
4 5
4 U
3 5 X
4 5
4 4Lxj
1 dtU7 DWrbI2
4 dtU7
1 3Y_Jo K-eol
1 n Uu--Z0o
4 n
1 TBp JUZGcZ4hO
1 xhJDE a
1 S imAkGjY
2 xhJDE
4 xhJDE
```

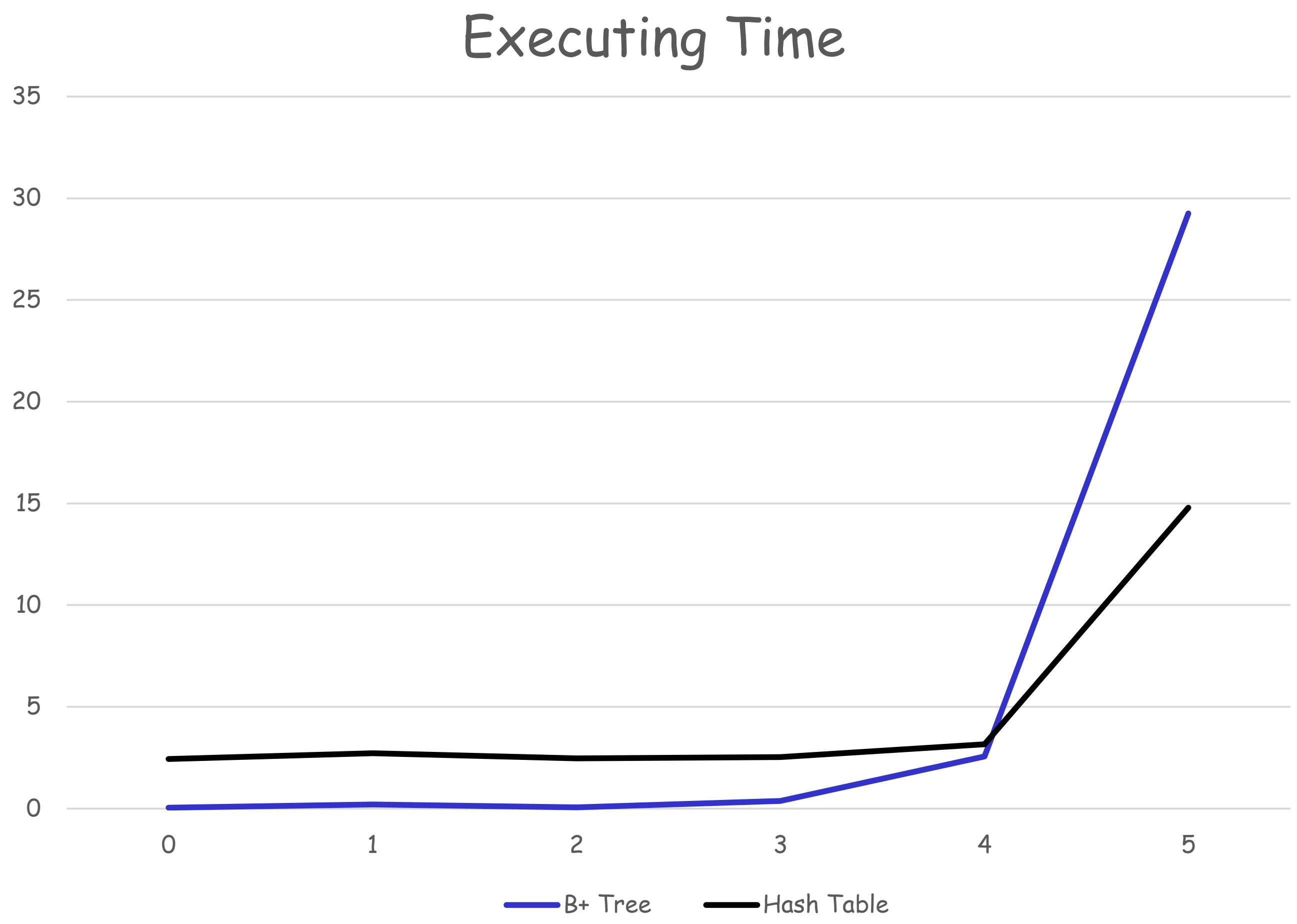
Test

- checker.cpp
  - Memory version using std::map
  - use for checking correctness

Comparison

size	B+ Tree	Hash Table
100	0.033s	2.429s
1000	0.204s	2.724s
1000	0.052s	2.462s
10000	0.378s	2.523s
100000	2.565s	3.165s
500000	29.252s	14.787s
2000000	36min42s	19min25s

Comparison: Time

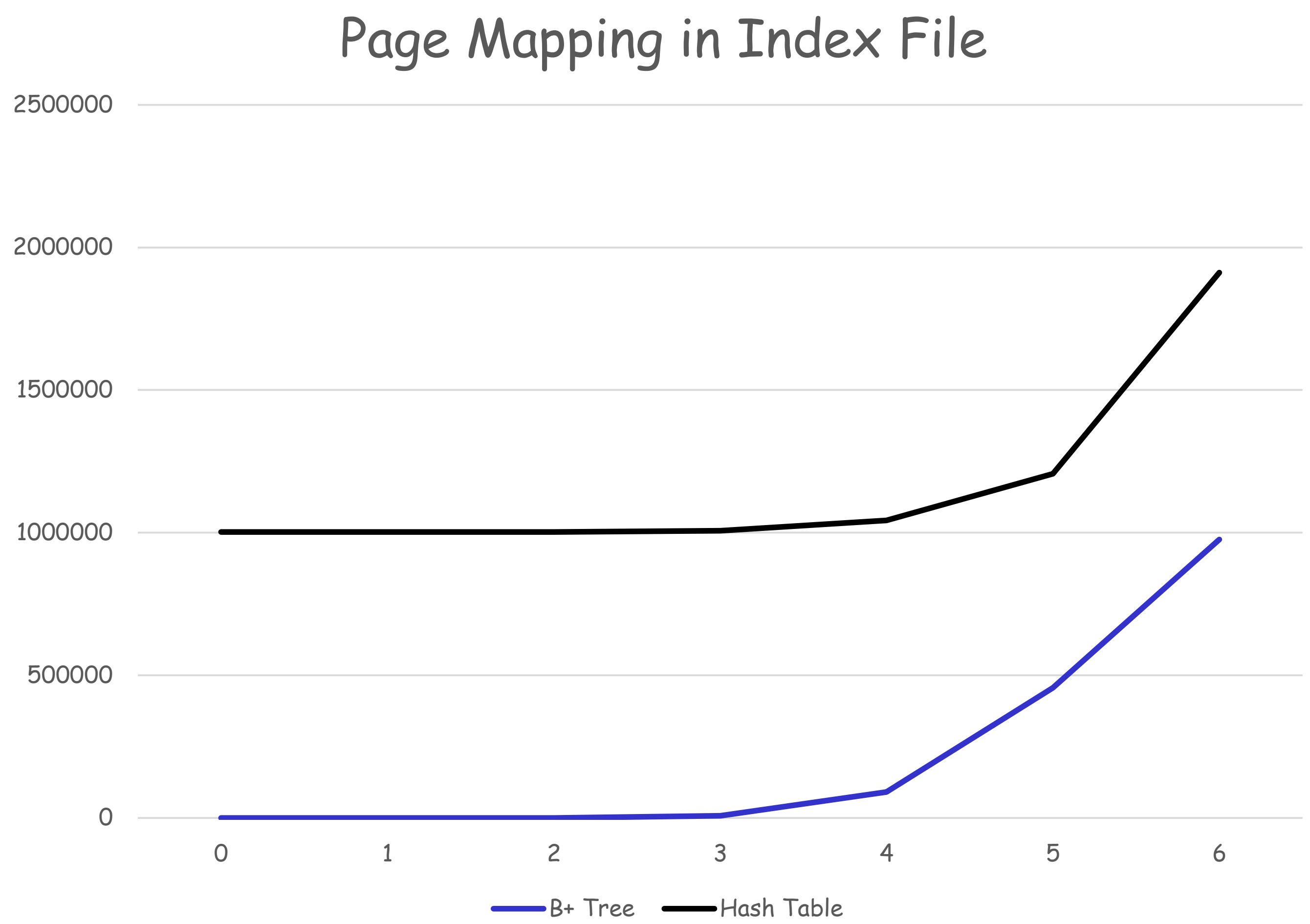


Time is not everything

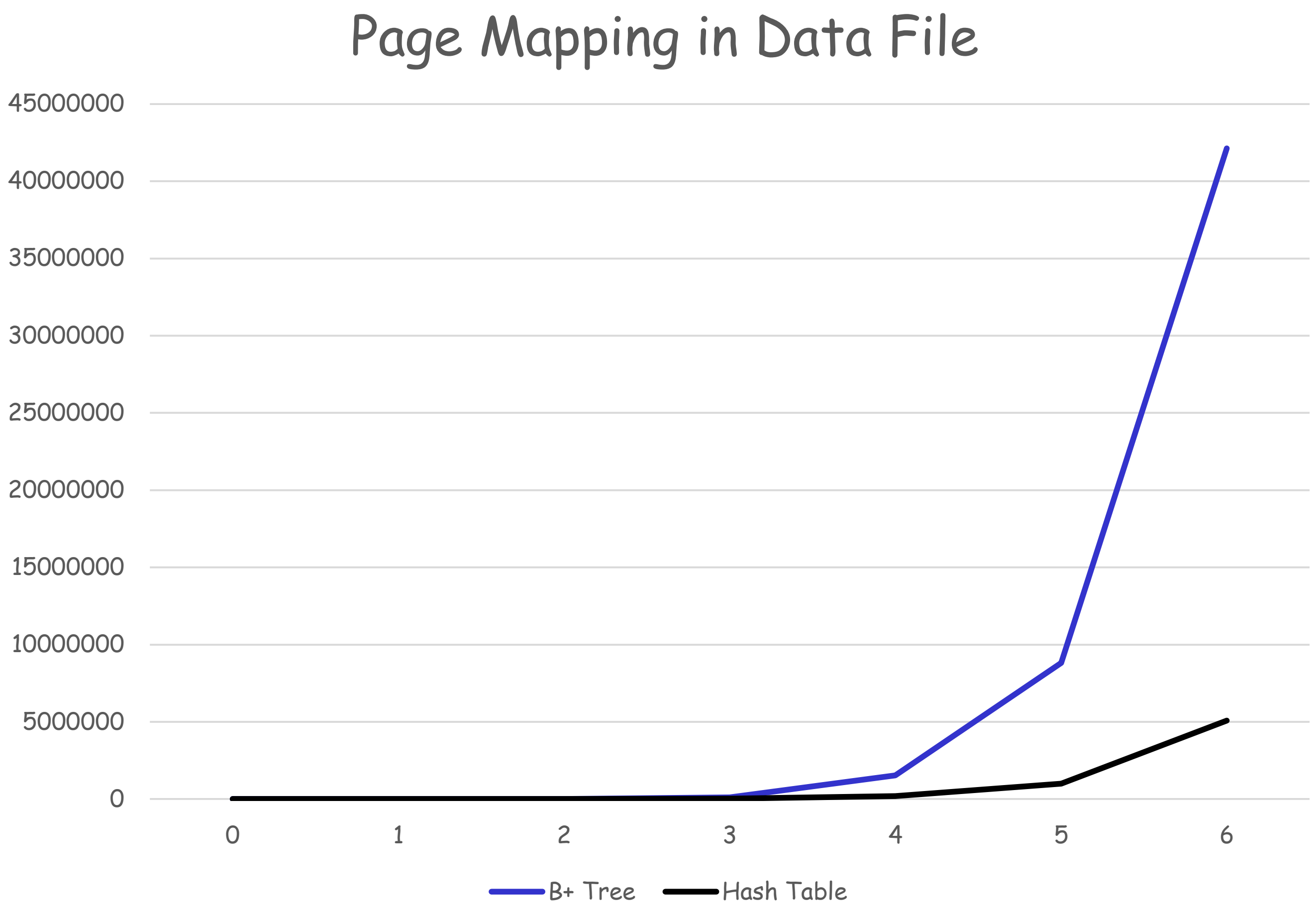
- How many times of page mapping does the two data structures cause in **Index File**?

- How many times of page mapping does the two data structures cause in **Data File**?

Comparison: Page Mapping



Comparison: Page Mapping

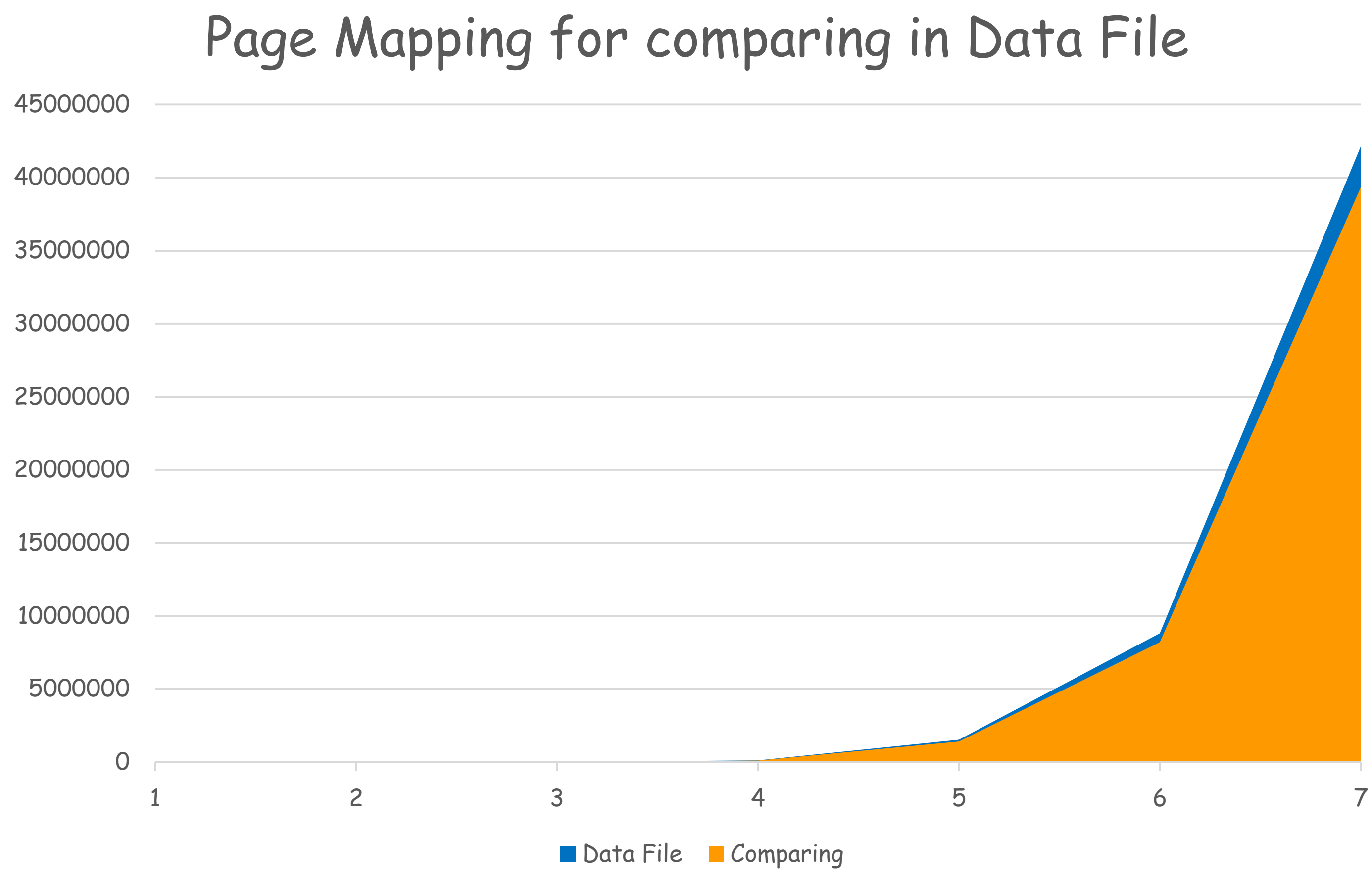


Comparison: B+ Tree

- Why B+ Tree needs so many page mapping in data file?

- Comparing keys !

Comparison: B+ Tree



Conclusion

- Hash Table is good when keys are arbitrary strings.
- B+ Tree is good when the complexity of comparing keys is small

Conclusion

- Databases in real world:
  - mysql: PRIMARY KEY is usually a number

- B+ Tree can compare keys with Index File only! Decrease the complexity of accessing Data File.

Thank You for listening